MATLAB Programming Style Requirements

Prepared by Peter Gagarinov

Moscow State University by M.V. Lomonosov

Faculty of Applied Mathematics and Computer Science

System Analysis Department

12 October, 2012

1 Introduction

This document is mainly based on [1] with some modifications added that make sense in the area of collaboration programming, creation of large MATLAB-based/scientifically intensive applications, large sharable libraries of /C++ functions. Most of the changes came from personal experience as MATLAB programmer.

2 Naming Conventions

In contrast to [1] naming convention described in this paper is more detailed as it is required for creation of sharable code.

2.1 Variables

The names of variables should document their meaning or use.

2.1.1 Variable names should be in mixed case starting with lower case.

This is common practice in the C++ development community. *MathWorks* sometimes starts variable names with upper case, but that usage is commonly reserved for types or structures in other languages. `midPrice,meanDev,indAsset,isAssetExpired`

An alternative technique is to use underscore to separate parts of a compound variable name. This technique, although readable, is not commonly used for variable names in other languages and is not recommended by the author.

2.1.2 Variables with a large scope should have more meaningful names. Variables with a small scope can have short but still meaningful names.

In practice most variables should have meaningful names. The use of short names should be reserved for conditions where they clarify the structure of the statements. Scratch variables used for temporary storage or indices can be kept short but still meaningful, like `ind,isAsset` instead of `indPair,isAssetExpired`. Using of common scratch variables `i,j,k,m,n` for integers and `x,y` for doubles is not recommended, especially considering the fact that `i,j` are the constants in MATLAB

2.1.3 The prefix n should be used for variables representing the number of objects.

This notation is taken from mathematics where it is an established convention for indicating the number of objects. Example: `nFiles,nSegments,nAssets,nCols`. A MATLAB-specific addition is the use of m for number of rows (based on matrix notation), as in `mRows`.

2.1.4 A convention on pluralization should be followed consistently.

A suggested practice is to make all variable names either singular or plural. Having two variables like `nDays,nDay,nAsset,nAssets` with names differing only by a final letter s should be avoided.

2.1.5 A convention on dimensionality should be followed consistently.

Use the suffices for indicating the dimensionality of the variables, namely `Vec(CVec)` for numeric (cell) vectors `Mat(CMat)` for numeric (cell) matrices, `Array(CArray)` for multi-dimentional numeric (cell) arrays: `point,pointMat` where `point` is singular and `pointMat` is plural.

2.1.6 Iterator variables should be prefixed with `i, j, k` etc.

The notation is taken from mathematics where it is an established convention for indicating iterators.

```
for iFile = 1:nFiles
    ...
end
```

For nested loops the iterator variables also should have helpful names.

```
%start cycle along assets
for iAssets = 1:nAssets
    %start cycle along metrics
    for jMetrics = 1:nMetrics
        ...
    end
end
```

2.1.7 Logical matrices should be prefixed with `is`(`isn`),index matrices should be prefixed with `ind`

Most of indexing job can be done by using either integer indeces or logical indeces. In order to distinguish such variables being used in the same part of the code use names like `indAssetExpired, indMaxMaturity, indMaxElem` for arrays of integer indeces and names like `isAssetExpired`,`isMaxMaturity`,`isMaxElem` for logical arrays. In that way it is possible to use both versions (logical and integer) simultaneously. For negated logical variables tend to use prefix `isn` instead of `isNot` as it make the code more compact: `isnMaxElem, isnNan`.

Indexation example

```
%initial parameters
nCols=10;
nRows=11;
%simulate data
randMat=rand(nRows,nCols);
%calculate indeces
isPositive=randMat>0; indPositive=find(isPositive);
%calculate number of positive elements: method #1
nPositive=sum(isPositive);
```

```
%%calculate number of positive elements: method #2
nPositiveAlt=numel(indPositive);
```

2.1.8 Using of negated boolean variable names is acceptable when it simplifies the code.

A contradictory situation arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what isnFound means. Use isFound and avoid isnFound in such cases but remember that there are many situations when use of negated logical variables is justified:

Negated logical variables

```
%initial parameters
nRows=15;
nCols=10;
%generate random matrix
randMat=rand(nRows,nCols);
%find negative elements
isnNeg=randMat>=0;
%find sum of negative elements
res=sum(randMat(isnNeg));
%display results
disp(res);
```

2.1.9 Acronyms, even if normally uppercase, should be mixed or lower case.

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable. When the name is connected to another, the readability is seriously reduced; the word following the abbreviation does not stand out as it should. Use html, isUsaSpecific. Avoid hTML, isUSASpecific.

2.1.10 Avoid using a keyword or special value name for a variable name.

MATLAB can produce cryptic error messages or strange results if any of its reserved words or builtin special values is redefined. Reserved words are listed by the command iskeyword. Special values are listed in the documentation.

2.2 Structures

2.2.1 Structure names should begin with a capital letter.

This usage is consistent with C++ practice, and it helps to distinguish between structures and ordinary variables.

2.2.2 The name of the structure is implicit, and need not be included in a field name.

Repetition is superfluous in use, as shown in the example. Use `Segment.length`. Avoid `Segment.segmentLength`.

2.2.3 Names of structure fields should begin with a lowercase letter even if the fields is a structure

The problem arise when the nesting of the structure is high and it can be annoying to use capital letter for all field-structures. Use `Data.dConf.calc.windowSize`. Avoid `Data.Conf.Calc.windowSize`.

2.3 Functions/class method names

The names of functions should document their use.

2.3.1 Names of functions should be written in lower case without underscores.

It is clearest to have the function and its m-file names the same. Using lower case avoids potential filename problems in mixed operating system environments. `getname(.)`, `computetotalwidth(.)`.

2.3.2 Functions should have meaningful names.

There is an unfortunate MATLAB tradition of using short and often somewhat cryptic function names probably due to the DOS 8 character limit. This concern is no longer relevant and the tradition should usually be avoided to improve readability. Use `computetotalwidth()`. Avoid `compwid()`. An exception is the use of abbreviations or acronyms widely used in mathematics. `max(.), gcd(.)`. Functions with such short names should always have the complete words in the first header comment line for clarity and to support `lookfor` searches.

2.3.3 Functions with a single output can be named for the output.

This is common practice in *MathWorks* code. `mean(.), standarderror(.)`

2.3.4 Functions with no output argument or which only return a handle should be named after what they do.

This practice increases readability, making it clear what the function should (and possibly should not) do. This makes it easier to keep the code clean of unintended side effects. `plot(.)`

2.3.5 The prefixes get/set should generally be reserved for accessing an object or property, not for functions

General practice of *MathWorks* and common practice in C++ and Java development. A plausible exception is the use of set for logical set operations. `getObj(.); setAppData(.)`

2.3.6 The prefix compute can be used in methods where something is computed.

Consistent use of the term enhances readability. Give the reader the immediate clue that this is a potentially complex or time consuming operation. `computeweightedaverage(); computespread()`.

2.3.7 The method names should consist of subwords each starting with a capital letter

Please note that the convention for object methods is different and every next subword should start with a capital letter. `doSomething(.); calculateAppData(.)`

2.3.8 The prefix find can be used in methods where something is looked up.

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability and it is a good substitute for get. `findOldestRecord(.); findHeaviestElement(.);`

2.3.9 The prefix is should be used for boolean functions.

Common practice in *MathWorks* code as well as C++ and Java. `isoverpriced(.); iscomplete(.)`

2.3.10 Complement names should be used for complement operations.

Reduce complexity by symmetry.

`get/set`, `add/remove`,`create/destroy`, `start/stop`, `insert/delete`, `increment/decrement`, `old/new`, `begin/end`, `first/last`, `up/down`, `min/max`, `next/previous`, `old/new`, `open/close`, `show/hide`, `suspend/resume`

2.3.11 Avoid unintentional shadowing.

In general function names should be unique. Shadowing (having two or more functions with the same name) increases the possibility of unexpected behavior or error. Names can be checked for shadowing using `which -all` or `exist`.

2.4 Classes

The names of classes should document their use.

2.4.1 Classes names should start with a capital letter.

`StateConfigurator` is just fine while `stateConfigurator` should be avoided.

2.4.2 Interfaces should with I.

`IStateConfigurator` is ok.

2.4.3 Abstract classes should with A.

`AStateConfigurator` is ok.

2.4.4 Classes names should not contain their full package names unless but can contain a small part of package name.

`AStateConfigurator` located within a package `modgen.configurators` is ok for two reasons: a) the package can contain the classes that are not configurators and b) When import instruction is used the package name can be omitted when referring to the class names and in this case `Configurator` suffix improves readability. `AModGenConfigurator` should be avoided.

2.5 Scripts

The names of scripts should document their use

2.5.1 Names of scripts should be prefixed with **s_** and should be written in lower case with underscores.

Use `s_gen_config`. Avoid `sgenconfig`,`sGenConfing`.

2.5.2 Prefer classes or functions to the scripts

In large applications a lot of scripts can produce a mess so because of all of the having the same variable visibility context. Use functions or classes instead

2.6 General

2.6.1 Names of constants should always be in upper-case

`MAX_ELEM_NUMBER`.

2.6.2 Names of dimensioned variables should usually have a units suffix.

Using a single set of units is an attractive idea that is only rarely implemented completely. Adding units suffixes helps to avoid the almost inevitable mixes. `positionQuotationUnits` or `positionQU`.

2.6.3 Abbreviations in names should be avoided.

Using whole words reduces ambiguity and helps to make the code self-documenting. Use `computearrivaltime(.)`. Avoid `comparr(.)`. Domain specific phrases that are more naturally known through their abbreviations or acronyms should be kept abbreviated. Even these cases might benefit from a defining comment near their first appearance. `std, var, html, cpu`.

2.6.4 Consider making names pronounceable.

Names that are at least somewhat pronounceable are easier to read and remember.

2.6.5 All names should be written in English.

The MATLAB distribution is written in English, and English is the preferred language for international development.

3 Files and Organization

Structuring code, both among and within files is essential to making it understandable. Thoughtful partitioning and ordering increase the value of the code.

The main difference between script and function is that normally the first one is not supposed to be reusable in other applications while the function is supposed to be logically solid peace of the code which is maintainable and reusable.

3.1 Functions

3.1.1 Modularize and partition

The best way to write a big program is to assemble it from well designed functions. This approach enhances readability, understanding and testing by reducing the amount of text which must be read to see what the code is doing. Code longer than two editor screens is a candidate for partitioning. Small well designed functions are more likely to be usable in other applications. All subfunctions and many functions should do one thing very well. Every function should hide something.

3.1.2 Do not use global variables for function interaction

The use of arguments is almost always clearer than the use of global variables. Moreover it is not correct when a function knows anything about a workspace surrounding it since this is a prerogative of the scripts to know exactly where their inputs and outputs are located.

3.1.3 Use common function interface format

Common function interface is a very important detail for development of large applications/toolboxes containing many functions. Here is proposed requirements for function interface

- Pass some part of input arguments as the properties if the number of arguments is greater than 3 so that the number of regular inputs does not exceed 3.
- Use the fixed list of regular arguments.

```
%Avoid writing the function which allows the following calls
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec);
%in that case windowSize is not a property
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec,...
    windowSize);
```

- Number of output arguments can be optional

```
%Matrix nAggregatedObsMat is an optional output
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec);
[dataOutMat,timeOutVec,nAggregatedObsMat]=...
    aggregatetimeseries(dataInpMat,timeInpVec,windowSize);
```

- Make all properties optional and define default values for all of them.

```
%Use window size = 15 lags (default value)
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec);
%Use specified windowSize=10 lags
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec,...
    'windowSize',10);

```

- Pass string inputs as the properties, names of the function regimes/methods in particular.

```
[dataOutMat,timeOutVec]=aggregatetimeseries(...
dataInpMat,timeInpVec,'method','simple','windowSize',1);
```

- Do not pass flags specifying methods and other function regimes as numbers. Use strings instead.

```
%Use
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec,...
    'method','advanced');
%Avoid
[dataOutMat,timeOutVec]=aggregatetimeseries(dataInpMat,timeInpVec,...
    'method',1);
```

- Pack output arguments into structure when the number of output arguments is greater than 3-5.

```
ResStruct=multipleoutputfunction(dataInpMat,timeInpVec,...
'method','simple','windowSize',1);
>> disp(ResStruct)
       outMat1: [30x40 double]
       outMat2: [30x40 double]
       outMat3: [10x40 double]
        method: 'simple'
```

Common function interface example

```
function [tsOut,tOut]=filterts(tsInp,tInp,varargin)
% FILTERTS filters time series according to selected methodId
%
% Usage: [tsOut,tOut]=filterts(tsInp,iInp,proplist)
%                    [tsOut,tOut]=filterts(tsInp,iInp,isSecondSkip,proplist)
%
% Input:
%         regular:
%           tsInp: double[nObs,nSeries] - input observed data,
%       tInp: double[nObs,1] - observed time,
%        optional:
%                isSecondSkip: logical[1,1] - if true, the second
%                        argument is skipped
%
%   properties:
%       methodId: string, specified the name of methodId,
%          'dummy' - simple methodId (tsOut=tsInp,tOut=tInp)
%
% Output:
%     regular:
%         tsOut: double[n,nSeries] - filtered time series,
%         tOut: double[n,1] - corresponding time
%
% Created by <Name> <FamilyName>, <University/Company>
```

3.1.4 Use existing functions.

Developing a function that is correct, readable and reasonably flexible can be a significant task. It may be quicker or surer to find an existing function that provides some or all of the required functionality.

3.1.5 Use function syntax instead of script syntax and operator syntax.

That way it is easy to control input parameters and avoid unpleasant errors.

Use `clear('a','b','c');` Avoid `clear a b c;`

3.1.6 Use function calls for hardly noticeable/distinguishable operators.

That approach significantly improves readability. Use `transpose(a);ctranspose(b);` Avoid `a.';a';`

3.1.7 Any block of code appearing in more than one function or script should be considered for packaging as a function.

It is much easier to manage changes if code appears in only one file. "Change is inevitable except from vending machines."

3.1.8 Subfunctions

A function used by only one other function should be packaged as its subfunction in the same file. This makes the code easier to understand and maintain.

4 Statements

4.1 Variables

4.1.1 Variables should not be reused unless required by memory limitation.

Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error from misunderstood definition.

4.2 Loops

4.2.1 Loop variables should be initialized immediately before the loop.

This improves loop speed and helps prevent bogus values if the loop does not execute for all possible indices.

```
1 resultMat = zeros(nEntries,1);
2 for iEntity = 1:nEntries
3     resultMat(iEntity)=foo(iEntity);
4 end
```

4.2.2 The end lines in nested loops can have comments

Adding comments at the end lines of long nested loops can help clarify which statements are in which loops and what tasks have been performed at these points.

4.3 Conditionals

4.3.1 Complex conditional expressions should be avoided.

Introduce temporary logical variables instead. By assigning logical variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug.

```
if (value>=lowerLimit)&(value<=upperLimit)&~ismember(value,valueArray)
    ...
end
```

should be replaced by:

```
isValid = (value >=lowerLimit) & (value <= upperLimit);
isNew = ~ismember(value,valueArray);
if (isValid & isNew)
    ...
end
```

4.3.2 The usual case should be put in the `if`-part and the exception in the `else`-part of an `if else` statement.

This practice improves readability by preventing exceptions from obscuring the normal path of execution.

```
fid = fopen(fileName);
if (fid~=-1)
 ...
else
 ...
end
```

4.3.3 The conditional expression `if 0` should be avoided, except for temporary block commenting

Make sure that the exceptions don't obscure the normal path of execution. Using the block comment feature of the editor is preferred.

4.3.4 A switch statement should include the otherwise condition

Leaving the otherwise out is a common error, which can lead to unexpected results.

```
switch (condition)
    case ABC,
        ...
    case DEF,
        ...
    otherwise
        ...
end
```

4.3.5 The `switch` variable should usually be a string.

Character strings work well in this context and they are usually more meaningful than enumerated cases.

4.4 General

4.4.1 Avoid cryptic code.

There is a tendency among some programmers, perhaps inspired by Shakespeare's line: Brevity is the soul of with, to write MATLAB code that is terse and even obscure. Writing concise code can be a way to explore the features of the language. However, in almost every circumstance, clarity should be the goal. As Steve Lord of *MathWorks* has written, "A month from now, if I look at this code, will I understand what it's doing? Try to avoid the situation described by the Captain in Cool Hand Luke, What we've got here is failure to communicate." The importance of this issue is underlined by many authors. Martin Fowler: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Kreitzberg and Shneiderman: "Programming can be fun, so can cryptography; however they should not be combined."

4.4.2 Use parentheses.

MATLAB has documented rules for operator precedence, but who wants to remember the details? If there might be any doubt, use parentheses to clarify expressions. They are particularly helpful for extended logical expressions.

4.4.3 The use of numbers in expressions should be minimized.

Numbers that are subject to change usually should be named constants instead. If a number does not have an obvious meaning by itself, readability is enhanced by introducing a named constant instead. It can be much easier to change the definition of a constant than to find and change all of the relevant occurrences of a literal number in a file.

4.4.4 Floating point comparisons should be made with caution.

Binary representation can cause trouble, as seen in this example.

```
1  %initialize input parameters with integer values
2  shortSide = 3;
3  longSide = 5;
4  otherSide = 4;
5  %make comparison in first way
6  longSide^2 ==(shortSide^2 + otherSide^2)
7  >> ans = 1
8  %initialize input parameters with real values
9  scaleFactor = 0.01;
10 %make comparison
11 (scaleFactor*longSide)^2 == ((scaleFactor*shortSide)^2 +...
```

```
12 (scaleFactor*otherSide)^2)
13 >> ans = 0
```

5 Layout, Comments and Documentation

5.1 Layout

The purpose of layout is to help the reader understand the code. Indentation is particularly helpful for revealing structure.

5.1.1 Content should be kept within the first 80 columns.

80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. Readability improves if unintentional line breaks are avoided when passing a file between programmers.

5.1.2 Lines should be split at graceful points.

Split lines occur when a statement exceeds the suggested 80 column limit. In general: " Break after a comma or space." Break after an operator. " Align the new line with the beginning of the expression on the previous line.

```
1 totalSum=a+b+c+...
2     d+e;
3 function(param1,param2,...
4     param3)
5 setText(['Long line split',...
6     'into two parts.']);
```

5.1.3 Indentation should be consistent with the MATLAB Editor.

The MATLAB editor provides indentation that clarifies code structure and is consistent with recommended practices for C++ and Java.

5.1.4 In general a line of code should contain only one executable statement.

This practice improves readability and allows JIT acceleration.

5.1.5 Logical groups of statements within a block should be separated by one blank line started with %.

Enhance readability by introducing comment symbols between logical units of a block.

5.1.6 Use cell divider %% for block separation. Add to this two lines started with %.

Cell divider %% while helps reading the code also makes additional features available in MATLAB Editor.

Separation example

```
%% Initial parameters
%
nCols=10;
nRows=10;
%
simMethod='norm';
%
%% Simulation
%
simMat=random(simMethod,nRows,nCols);
%
%% Display results
%
disp(simMethod);
disp(simMat);
```

5.2 Comments

The purpose of comments is to add information to the code.

Typical uses for comments are to explain usage, provide reference information, to justify decisions, to describe limitations, to mention needed improvements. Experience indicates that it is better to write comments at the same time as the code rather than to intend to add comments later.

5.2.1 Comments cannot justify poorly written code.

Comments cannot make up for code lacking appropriate name choices and an explicit logical structure. Such code should be rewritten. Steve McConnell: "Improve the code and then document it to make it even clearer."

5.2.2 Comments should agree with the code, but do more than just restate the code.

A bad or useless comment just gets in the way of the reader. N. Schryer: "If the code and the comments disagree, then both are probably wrong." It is usually more important for the comment to address why or how rather than what.

5.2.3 Comments should be easy to read.

There should be a space between the upper case letter and end with a period.

5.2.4 Comments should usually have the same indentation as the statements referred to.

This is to avoid having the comments break the layout of the program. End of line comments tend to be cryptic and should be avoided except for constant definitions.

5.2.5 Use common function header format.

Stick with the following format of function header.

Function header format

```
% <FunctionNameInUpperCase> short functionality description
%
% detailed description
%
% Usage: usage format (all usage scenarious should be listed)
%
% input:
%     regular:
%         <inpArgName1>: <type> <dimension> - description
%           ...
%         <inpArgNameN>: <type> <dimension> - description
%     properties:
%         <propName1>:<type> - description
%           ...
%         <propNameK>:<type> - description
%
% output:
%     regular:
%         <outArgName1>: <type> <dimension> - description
%           ...
%         <outArgNameM>: <type> <dimension> - description
%
% Example: one or more usage examples

% Created by <AuthorName(s)>, <CopyRightDescription>

```

Note that

- Function header comments should support the use of help and lookfor.

  `help` prints the first contiguous block of comment lines from the file. Make it helpful. `lookfor` searches the first comment line of all m-files on the path. Try to include likely search words in this line.

- Function header comments should discuss any special requirements for the input arguments.

  The user will need to know if the input needs to be expressed in particular units

```
% ejectionFraction must be between 0 and 1, not a percentage.
% elapsedTimeSeconds must be one dimensional.
```

- Function header comments should describe any side effects.

  Side effects are actions of a function other than assignment of the output variables. A common example is plot generation. Descriptions of these side effects should be included in the header comments so that they appear in the help printout.

- One should avoid clutter in the help printout of the function header.

  It is common to include copyright lines and change history in comments near the beginning of a function file. There should be a blank line between the header comments and these comments so that they are not displayed in response to help.

5.2.6 Use common script header format

The outputs and inputs of MATLAB script are usually harder to specify. However it is very helpful to stick with common script header format.

Script header format

```
% <ScriptNameInUpperCase> short functionality description
%
% detailed description
%
% Input:
%   description of inputs (what variables have to be set before
%       running the script)
%
% Output:
%   description of outputs (what variables takes values after
%       execution of the script)
%
% Created by <AuthorName(s)>, <CompanyName>

```

5.2.7 All comments should be written in English.

In an international environment, English is the preferred language.

5.3 Documentation

5.3.1 Formal documentation

To be useful documentation should include a readable description of what the code is supposed to do (Requirements), how it works (Design), which functions it depends on and how it is used by other code (Interfaces), and how it is tested. For extra credit, the documentation can include a discussion of alternative solutions and suggestions for

extensions or maintenance. Dick Brandon: "Documentation is like sex; when it's good, it's very, very good, and when it's bad, it's better than nothing."

5.3.2 Consider writing the documentation first

Some programmers believe that the best approach is "Code first and answer questions later." Through experience most of us learn that developing a design and then implementing it leads to a much more satisfactory result. Development projects are almost never completed on schedule. If documentation and testing are left for last, they will get cut short. Writing the documentation first assures that it gets done and will probably reduce development time.

5.3.3 Changes.

The professional way to manage and document code changes is to use a source control tool. For very simple projects, adding change history comments to the function files is certainly better than nothing.

References

[1] *Richard Johnson*, MATLAB Programming Style Guidelines.